Cave à vins (Partie 13)

# TP02 du module 06 – Sécurité

|  |
| --- |
| Ce TP sera un TP fil rouge du cours   * Transfert d’un jeton JWT entre le client et le serveur * Mise en place des librairies JJWT * Création de la configuration associée |

|  |
| --- |
| **Durée estimée** |
| 1 heure 30 |

Contexte

En vous appuyant sur l’ensemble des concepts abordés dans le cours, mettez en place une application Spring Boot permettant la gestion d’une cave à vins.

L’application de cave à vin sera une application d’API Web sécurisée à la fin des itérations :

* Elle permet de gérer le stock des bouteilles de vins par le propriétaire qui est un administrateur de l’application
* Elle permet de gérer l’achat en ligne des bouteilles par un client
  + Les bouteilles de vin peuvent avoir 3 couleurs : Rouge, Blanc ou Rosé
  + Les bouteilles de vin sont associées aux régions françaises
* Un client et un propriétaire doivent s’authentifier
* Un visiteur est un client qui n’a pas de compte
* Il pourra uniquement consulter le stock

Dans les itérations nous réaliserons uniquement le back de l’application.

Cette application se fera avec Spring Boot.

Dans les itérations de ce module, nous allons sécuriser notre API Web.

Solution

Pour réaliser ce TP, il faut vous aider de la démonstration et des schémas du cours

* Cycle d’authentification

Une image contenant texte, capture d’écran, Police, diagramme

Description générée automatiquement

* Cycle d’autorisation

Une image contenant texte, Police, capture d’écran, diagramme

Description générée automatiquement

# Librairies JJWT :

Il faut ajouter les lib librairies JJWT API et JJWT Impl pour permettre de manipuler le jeton JWT

* Dans build.gradle ; ajouter les librairies pour JWT

// https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api

**implementation** 'io.jsonwebtoken:jjwt-api:0.11.5'

// https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl

**runtimeOnly** 'io.jsonwebtoken:jjwt-impl:0.11.5'

// https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-jackson

**runtimeOnly** 'io.jsonwebtoken:jjwt-jackson:0.11.5'

# L’entité Utilisateur :

Dans les 2 schémas, il faut manipuler les données d’un utilisateur depuis la base. Pour cela, il faut une entité et un Repository associé

* L’entité correspond à la table « cav\_user » déjà existante

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

## L’entité Utilisateur doit implémenter l’interface UserDetails

Pour manipuler cette entité dans le contexte de Spring Security, il faut que l’entité implémente UserDetails

* Cette interface appartient à Spring Security
  + Elle représente l’utilisateur a connecté pour Spring Security
* Pour comprendre cette classe et ce qu’elle implique
  + Il faut aller explorer la classe : org.springframework.security.core.userdetails.UserDetails
  + https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/userdetails/UserDetails.html

**package** fr.eni.cave.bo;

**import** java.util.\*;

**import** org.springframework.security.core.GrantedAuthority;

**import** org.springframework.security.core.authority.SimpleGrantedAuthority;

**import** org.springframework.security.core.userdetails.UserDetails;

**import** jakarta.persistence.\*;

**import** lombok.\*;

**import** lombok.experimental.SuperBuilder;

@NoArgsConstructor

@AllArgsConstructor

@Getter

@Setter

@EqualsAndHashCode(of = { "pseudo" })

@ToString(of = { "pseudo", "nom", "prenom" })

@SuperBuilder

@Entity

@Table(name = "CAV\_USER")

@Inheritance(strategy = InheritanceType.***JOINED***)

**public** **class** Utilisateur **implements** UserDetails{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Id

@Column(name = "LOGIN", nullable = **false**, length = 255)

**private** String pseudo;

@Column(name = "PASSWORD", nullable = **false**, length = 68)

**private** String password;

@Column(name = "LAST\_NAME", nullable = **false**, length = 90)

**private** String nom;

@Column(name = "FIRST\_NAME", nullable = **false**, length = 150)

**private** String prenom;

@Column(length = 15)

**private** String authority;

/\*\*

\* Méthodes associées à UserDetails

\*/

@Override

**public** Collection<? **extends** GrantedAuthority> getAuthorities() {

**return** Arrays.*asList*(**new** SimpleGrantedAuthority(authority));

}

@Override

**public** String getUsername() {

**return** pseudo;

}

@Override

**public** **boolean** isAccountNonExpired() {

**return** **true**;

}

@Override

**public** **boolean** isAccountNonLocked() {

**return** **true**;

}

@Override

**public** **boolean** isCredentialsNonExpired() {

**return** **true**;

}

@Override

**public** **boolean** isEnabled() {

**return** **true**;

}

}

* Notre classe mère hérite de UserDetails
  + Qui demande un numéro de sérialisation
  + Il faut aussi en ajouter un sur chaque classe fille

# UserDetailsService :

Pour manipuler UserInfo avec Spring Security ; il faut déclarer UserDetailsService

* UserDetailsService doit être déclaré dans une classe de configuration
* Création d’une classe JwtAppConfig
  + Elle centralise la configuration de sécurité liée à l’entité Utilisateur
  + Manipule la base de données au travers de UtilisateurRepository
  + Et encapsule la sécurité : AuthenticationProvider et AuthenticationManager

**package** fr.eni.cave.security.jwt;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.context.annotation.\*;

**import** org.springframework.security.authentication.\*;

**import** org.springframework.security.authentication.dao.DaoAuthenticationProvider;

**import** org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;

**import** org.springframework.security.core.userdetails.\*;

**import** fr.eni.cave.dal.UtilisateurRepository;

@Configuration

**public** **class** JwtAppConfig {

/\*\*

\* Authentification de l'utilisateur depuis la base de données

\*/

@Autowired

**private** UtilisateurRepository uRepository;

@Bean

UserDetailsService userDetailsService() {

**return** username -> uRepository.findById(username)

.orElseThrow(() -> **new** UsernameNotFoundException("User not found"));

}

@Bean

AuthenticationProvider authenticationProvider() {

DaoAuthenticationProvider authProvider = **new** DaoAuthenticationProvider();

authProvider.setUserDetailsService(userDetailsService());

**return** authProvider;

}

@Bean

AuthenticationManager authenticationManager(AuthenticationConfiguration config) **throws** Exception {

**return** config.getAuthenticationManager();

}

}

# Créer JwtService

D’après les 2 schémas, JwtService va gérer le jeton JWT.

* C’est pour manipuler le jeton JWT, qu’il fallait ajouter les dépendances JJWT API et JJWT Impl
* JwtService permet de
  + Générer un jeton
    - Définir l'entête,
    - Les « claims » « champs standards »,
    - Le délai d'expiration
    - Et permet de signer avec la « secret key »
  + Elle doit vérifier l'authenticité du jeton et obtenir les garanties de jeton
  + Elle définit une méthode extractUsername(String token) pour retourner le pseudo contenu dans le jeton
  + Déclarer une clef de chiffrement sur 256 en héxadécimal
    - La déclarer en paramètre dans application.properties

#JWT

app.jwt.secret=2529355127…………

* Créer la classe JwtService
  + Récupérer le paramètre dans la classe JwtService
  + Il faut aussi créer la méthode getSigningKey
  + La clef de signature est utilisée pour signer numériquement le JWT
  + Une « secret key » est utilisée pour créer la partie « signature » du JWT
  + Permet de vérifier que l’expéditeur du JWT est celui qu’il prétend
  + Et être sûr que votre message n’a pas été modifié en route
  + La clef de signature est utilisée conjointement à l’algorithme de connexion spécifié dans l’entête
  + hmacShaKeyFor est l’un des algorithmes de connexion précédemment cités
* Il faut des méthodes pour extraire du jeton tous les « claims » ou un particulier
* Ajouter une méthode pour extraire le pseudo de l’utilisateur
  + C’est une donnée du « subject »
* Ajouter 2 méthodes pour générer le jeton JWT
  + La première permet de préciser une durée de vie, la clef de signature et le contexte utilisateur à conserver
  + La deuxième permet de créer le jeton avec les données vides
* Ajouter des méthodes pour gérer la validation du jeton

**package** fr.eni.cave.security.jwt;

**import** java.security.Key;

**import** java.util.\*;

**import** java.util.function.Function;

**import** org.springframework.beans.factory.annotation.Value;

**import** org.springframework.security.core.userdetails.UserDetails;

**import** org.springframework.stereotype.Service;

**import** io.jsonwebtoken.\*;

**import** io.jsonwebtoken.io.Decoders;

**import** io.jsonwebtoken.security.Keys;

@Service

**public** **class** JwtService {

// Déclarer une clef de sécurité, en utilisant

@Value("${app.jwt.secret}")

**private** String SECRET\_KEY;

//Signature transmise pour la création du jeton.

//Et chiffrer-déchiffrer les données du jeton

**private** Key getSignInKey() {

**byte**[] keyBytes = Decoders.***BASE64***.decode(SECRET\_KEY);

**return** Keys.*hmacShaKeyFor*(keyBytes);

}

// Extraire « claims » du jeton

**private** Claims extractAllClaims(String token) {

**return** Jwts

.*parserBuilder*()

.setSigningKey(getSignInKey())

.build()

.parseClaimsJws(token)

.getBody();

}

// Extraire 1 « claims » du jeton

**private** <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {

**final** Claims claims = extractAllClaims(token);

**return** claimsResolver.apply(claims);

}

// Extraire le pseudo du jeton

**public** String extractUserName(String token) {

**return** extractClaim(token, Claims::getSubject);

}

// Générer le jeton JWT

**public** String generateToken(Map<String, Object> extraClaims, UserDetails userDetails) {

**return** Jwts

.*builder*()

.setClaims(extraClaims)

.setSubject(userDetails.getUsername())

.setIssuedAt(**new** Date(System.*currentTimeMillis*()))

.setExpiration(**new** Date(System.*currentTimeMillis*() + 1000 \* 60 \* 24))

.signWith(getSignInKey(), SignatureAlgorithm.***HS256***)

.compact();

}

**public** String generateToken(UserDetails userDetails) {

**return** generateToken(**new** HashMap<>(), userDetails);

}

// Validation du jeton

**public** **boolean** isTokenValid(String token, UserDetails userDetails) {

**final** String username = extractUserName(token);

**return** (username.equals(userDetails.getUsername()) && !isTokenExpired(token));

}

**private** **boolean** isTokenExpired(String token) {

**return** extractExpiration(token).before(**new** Date());

}

**private** Date extractExpiration(String token) {

**return** extractClaim(token, Claims::getExpiration);

}

}

Nous avons terminé de créer la classe JwtService qui permet de créer, manipuler et valider le jeton

# Créer JWTAuthenticationFilter

Si on regarde notre schéma, l’élément qui intercepte les requêtes est JwtAuthenticationFilter

* Elle hérite d’OncePerRequestFilter
  + Comme son nom l’indique ; il sera utilisé à chaque appel
* Elle doit manipuler 2 services
  + JwtService 🡪 qui gèrera le jeton JWT
  + UserDetailsService est une interface de Spring Security qui permet de manipuler l’utilisateur remonté de la base de données
* Création de JwtAuthFilter
  + Lui Injecter JwtService et UserDetailsService
* Implémenter la méthode de l’interface : doFilterInternal
  + HttpServletRequest permet de manipuler la requête http request
  + HttServletResponse permet de manipuler la réponse http
  + Le fait d’avoir accès au protocole http, cela nous permet d’intercepter chaque requête http et de manipuler le comportement
  + Il nous est possible de capturer des données, changer les en-têtes et renvoyer des données, …
  + FilterChain, représente la chaîne d’événements
  + Les 3 paramètres ne doivent pas être null
    - Il est possible d’ajouter l’annotation @NonNull
  + D’après le schéma, la première chose à vérifier dans JwtAuthFilter est le JWT Token
    - Le jeton JWT est passé dans l’entête de la requête
  + Sur la classe HttpRequest il existe la méthode getHeader pour récupérer les données
    - La clef dans l’entête s’appelle « Authorization »
    - Les jetons JWT commence par "Bearer "
    - Structure de l’entête : « Authorization: Bearer <jeton> »
    - Donc si dans la value de l’entête "Bearer " n’est pas présent c’est qu’il n’est pas correct
* Il faut terminer le processus de validation « Validate JWT » dans le filtre JwtAuthFilter
  + En appelant les méthodes de validation de JwtService
  + Et il faut aussi appeler la validation de DB, il faut pour cela utiliser UserDetailsService
  + Mise à jour du contexte de sécurité avec le jeton créé
  + Et pour finir la méthode permettre de transmettre le contexte du filtre

**package** fr.eni.cave.security.jwt;

**import** java.io.IOException;

**import** org.springframework.lang.NonNull;

**import** org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

**import** org.springframework.security.core.context.SecurityContextHolder;

**import** org.springframework.security.core.userdetails.\*;

**import** org.springframework.security.web.authentication.WebAuthenticationDetailsSource;

**import** org.springframework.stereotype.Component;

**import** org.springframework.web.filter.OncePerRequestFilter;

**import** jakarta.servlet.\*;

**import** jakarta.servlet.http.\*;

//Doit être active dès qu'il y a une requête

//Doit devenir un bean pour spring

@Component

**public** **class** JwtAuthenticationFilter **extends** OncePerRequestFilter {

// Injection de la couche BLL pour gérer le token

**private** JwtService jwtService;

// Injection de la couche BLL pour gérer les données de la DB

**private** UserDetailsService userDetailsService;

**public** JwtAuthenticationFilter(JwtService jwtService, UserDetailsService userDetailsService) {

**this**.jwtService = jwtService;

**this**.userDetailsService = userDetailsService;

}

**protected** **void** doFilterInternal(@NonNull HttpServletRequest request,

@NonNull HttpServletResponse response,

@NonNull FilterChain filterChain)

**throws** ServletException, IOException {

// vérifier le jeton JWT – il est passé dans l’en-tête

**final** String authHeader = request.getHeader("Authorization");

**final** String jwt;

// vérifier qu’il y a une donnée dans l’entête qui correspond à Authorization

// l’entête contient Bearer <jeton> SINON erreur

// Sinon laissé le comportement suivre son cours

**if** (authHeader == **null** || !authHeader.startsWith("Bearer ")) {

filterChain.doFilter(request, response);

**return**;

}

// Il y a un JWT – il faut l’extraire

jwt = authHeader.substring(7);// 7 correspond à Bearer

// Vérification de l'utilisateur

**final** String userEmail = jwtService.extractUserName(jwt);// Extraire du jeton JWT

// Validation des données par rapport à la DB

**if** (userEmail != **null** && SecurityContextHolder.*getContext*().getAuthentication() == **null**) {

// Check in DB

UserDetails userDetails = **this**.userDetailsService.loadUserByUsername(userEmail);

// Validation du jeton JWT

**if** (jwtService.isTokenValid(jwt, userDetails)) {

// Gestion du contexte de sécurité de l’utilisateur

// Création d'un nouveau jeton avec les informations et les rôles de

// l'utilisateur

UsernamePasswordAuthenticationToken authToken = **new**

UsernamePasswordAuthenticationToken(userEmail, **null**,

userDetails.getAuthorities());

// Transmettre les détails de la demande d’origine

authToken.setDetails(**new** WebAuthenticationDetailsSource()

.buildDetails(request));

// Mise à jour du contexte de sécurité

SecurityContextHolder.*getContext*().setAuthentication(authToken);

}

}

filterChain.doFilter(request, response);

}

}

# Contexte de Spring Security

D’après nos schémas :

* Activer le JwtAuthenticationFilter comme premier filtre avec celui du contexte de sécurité
* Définir la solution de reconnaissance de l’utilisateur à connecter (JwtAppConfig)
* Ajouter dans la classe AppConfigSecurity
  + Injection du filtre : JwtAuthenticationFilter
  + Injection d’AuthenticationProvider 🡪 il correspond au contexte de la classe JwtAppConfig
* Supprimer de la classe AppConfigSecurity ; la méthode userDetailsManager
  + Elle n’est plus utilisée
  + Elle est remplacée par AuthenticationManager et AuthenticationProvider
* Il faut ajouter un URL racine qui sera valide pour tout le monde pour l’authentification
* Dans la méthode filterChain
  + Remplacer l’authentification « httpBasic »
  + Par l’appel de notre bean authentificationProvider
  + Activation du filtre JWT
  + Nous voulons que la session soit Stateless, donc nous le forçons dans le filtre

**package** fr.eni.cave.security;

**import** org.apache.commons.logging.\*;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.context.annotation.\*;

**import** org.springframework.http.HttpMethod;

**import** org.springframework.security.authentication.\*;

**import** org.springframework.security.config.annotation.web.builders.HttpSecurity;

**import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

**import** org.springframework.security.config.http.SessionCreationPolicy;

**import** org.springframework.security.web.SecurityFilterChain;

**import** org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

**import** jakarta.servlet.Filter;

@Configuration

@EnableWebSecurity

**public** **class** AppConfigSecurity {

**protected** **final** Log logger = LogFactory.*getLog*(getClass());

@Autowired

**private** Filter jwtAuthenticationFilter;

@Autowired

**private** AuthenticationProvider authenticationProvider;

/\*\*

\* Restriction des URLs selon la connexion utilisateur et leurs rôles

\*/

@Bean

SecurityFilterChain filterChain(HttpSecurity http) **throws** Exception {

http.authorizeHttpRequests(auth -> {

auth

// Consulter le stock des bouteilles - est accessible à tout le monde

.requestMatchers("/caveavin/bouteilles").permitAll() .requestMatchers("/caveavin/auth/\*\*").permitAll()

// Permettre aux rôles CLIENT et OWNER de manipuler :

.requestMatchers(HttpMethod.***GET***, "/caveavin/paniers/\*\*").hasAnyRole("CLIENT", "OWNER")

.requestMatchers(HttpMethod.***GET***, "/caveavin/paniers/client/actifs/\*\*").hasAnyRole("CLIENT", "OWNER")

.requestMatchers(HttpMethod.***GET***, "/caveavin/paniers/client/commandes/\*\*").hasAnyRole("CLIENT", "OWNER")

// Restreindre au rôle CLIENT

.requestMatchers(HttpMethod.***POST***, "/caveavin/paniers").hasRole("CLIENT")

.requestMatchers(HttpMethod.***PUT***, "/caveavin/paniers").hasRole("CLIENT")

// Restreindre au rôle OWNER

.requestMatchers(HttpMethod.***GET***, "/caveavin/bouteilles/\*\*").hasRole("OWNER")

.requestMatchers(HttpMethod.***POST***, "/caveavin/bouteilles").hasRole("OWNER")

.requestMatchers(HttpMethod.***PUT***, "/caveavin/bouteilles").hasRole("OWNER")

.requestMatchers(HttpMethod.***DELETE***, "/caveavin/bouteilles").hasRole("OWNER")

// Toutes autres url et méthodes HTTP ne sont pas permises

.anyRequest().denyAll();

});

// Désactivé Cross Site Request Forgery

// Non préconisé pour les API REST en stateless.

// Sauf pour POST, PUT, PATCH et DELETE

http.csrf(csrf -> {

csrf.disable();

});

//Connexion de l'utilisateur

http.authenticationProvider(authenticationProvider);

//Activer le filtre JWT et l'authentication de l'utilisateur

http.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.**class**);

// Session Stateless

http.sessionManagement(session -> {

session.sessionCreationPolicy(SessionCreationPolicy.***STATELESS***);

});

**return** http.build();

}

}

# Contrôleur et service d’authentification

D’après notre schéma d’authentification, il nous reste à créer : AuthenticationController et AuthenticationService

* Le contrôleur AuthenticationController permettra de récupérer les données d’authentification
  + L’URL racine déclaré ; dans la classe AppConfigSecurity : /caveavin/auth/\*\*
  + Il faut qu’il manipule la base de données, au travers du service AuthenticationService
* Il déclare une méthode sur Post pour gérer l’authentification
* Pour cette méthode il faut déclarer
  + AuthenticationResponse 🡪 POJO contenant uniquement le jeton JWT à retourner
  + AuthenticationRequest 🡪 POJO représentant les données du formulaire de connexion
* Créer AuthenticationResponse et AuthenticationRequest

**package** fr.eni.cave.security.jwt;

**import** lombok.\*;

@Data

@AllArgsConstructor

@NoArgsConstructor

@EqualsAndHashCode(of = "pseudo")

**public** **class** AuthenticationRequest {

**private** String pseudo;

**private** String password;

}

**package** fr.eni.cave.security.jwt;

**import** lombok.\*;

@Data

@AllArgsConstructor

@NoArgsConstructor

@EqualsAndHashCode

**public** **class** AuthenticationResponse {

**private** String token;

}

* Créer AuthenticationService
  + Il permet de manipuler les données en base
    - Pour cela, il faut injecter notre UtilisateurRepository pour récupérer
  + D’authentifier un utilisateur
    - Pour cela, il faut injecter [org](eclipse-javadoc:%E2%98%82=demo-jwt2/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.springframework.security%5C/spring-security-core%5C/6.0.3%5C/d3b35481b651c777fd7366b8a496a1d67098fed3%5C/spring-security-core-6.0.3.jar=/gradle_used_by_scope=/main,test=/%3Corg).[springframework](eclipse-javadoc:%E2%98%82=demo-jwt2/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.springframework.security%5C/spring-security-core%5C/6.0.3%5C/d3b35481b651c777fd7366b8a496a1d67098fed3%5C/spring-security-core-6.0.3.jar=/gradle_used_by_scope=/main,test=/%3Corg.springframework).[security](eclipse-javadoc:%E2%98%82=demo-jwt2/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.springframework.security%5C/spring-security-core%5C/6.0.3%5C/d3b35481b651c777fd7366b8a496a1d67098fed3%5C/spring-security-core-6.0.3.jar=/gradle_used_by_scope=/main,test=/%3Corg.springframework.security).[authentication](eclipse-javadoc:%E2%98%82=demo-jwt2/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.springframework.security%5C/spring-security-core%5C/6.0.3%5C/d3b35481b651c777fd7366b8a496a1d67098fed3%5C/spring-security-core-6.0.3.jar=/gradle_used_by_scope=/main,test=/%3Corg.springframework.security.authentication).AuthenticationManager
    - L’utiliser pour authentifier l’utilisateur avec son pseudo et mot de passe
* authenticate 🡪 cette méthode retourne une exception si l’authentification échoue
  + Si OK 🡪 Récupérer l’utilisateur depuis notre UserRepository
  + Transmettre une exception si non trouvé avec orElseThrow()
  + Si tout est OK 🡪 Création d’un jeton JWT associé
    - Injecter JwtService pour accéder à la création d’un jeton JWT
    - Et injecter ce jeton dans l’AuthenticationResponse

**package** fr.eni.cave.security.jwt;

**import** org.springframework.security.authentication.\*;

**import** org.springframework.stereotype.Service;

**import** fr.eni.cave.bo.Utilisateur;

**import** fr.eni.cave.dal.UtilisateurRepository;

**import** lombok.AllArgsConstructor;

@AllArgsConstructor

@Service

**public** **class** AuthenticationService {

**private** UtilisateurRepository uRepository;

**private** AuthenticationManager authenticationManager;

**private** JwtService jwtService;

**public** AuthenticationResponse authenticate(AuthenticationRequest request) {

authenticationManager.authenticate(

**new** UsernamePasswordAuthenticationToken(

request.getPseudo(), request.getPassword()));

Utilisateur user = uRepository.findById(request.getPseudo()).orElseThrow();

String jwtToken = jwtService.generateToken(user);

AuthenticationResponse authResponse = **new** AuthenticationResponse();

authResponse.setToken(jwtToken);

**return** authResponse;

}

}

* Créer AuthenticationController

**package** fr.eni.cave.security.jwt;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/caveavin/auth")

**public** **class** AuthenticationController {

**private** **final** AuthenticationService authenticationService;

**public** AuthentificationController(AuthenticationService authenticationService) {

**this**.authenticationService = authenticationService;

}

@PostMapping

**public** ResponseEntity<AuthenticationResponse> register(@RequestBody AuthenticationRequest request) {

**return** ResponseEntity.*ok*(authenticationService.authenticate(request));

}

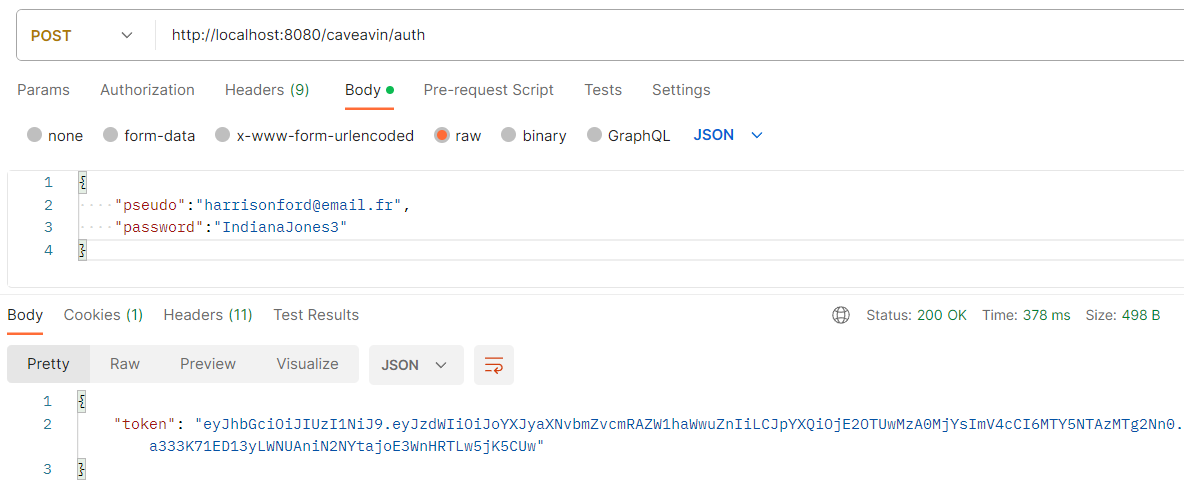
}

* Nous avons à présent l’ensemble des classes, beans de nos schémas

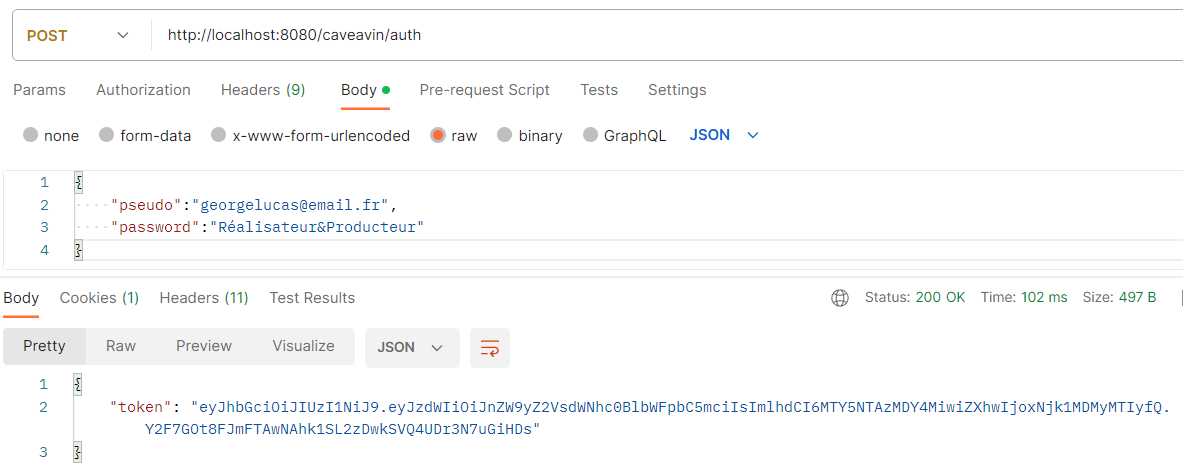
# Exécution :

Pour manipuler notre sécurité avec jeton JWT

1. Créer un jeton JWT en connectant un Utilisateur avec le rôle CLIENT

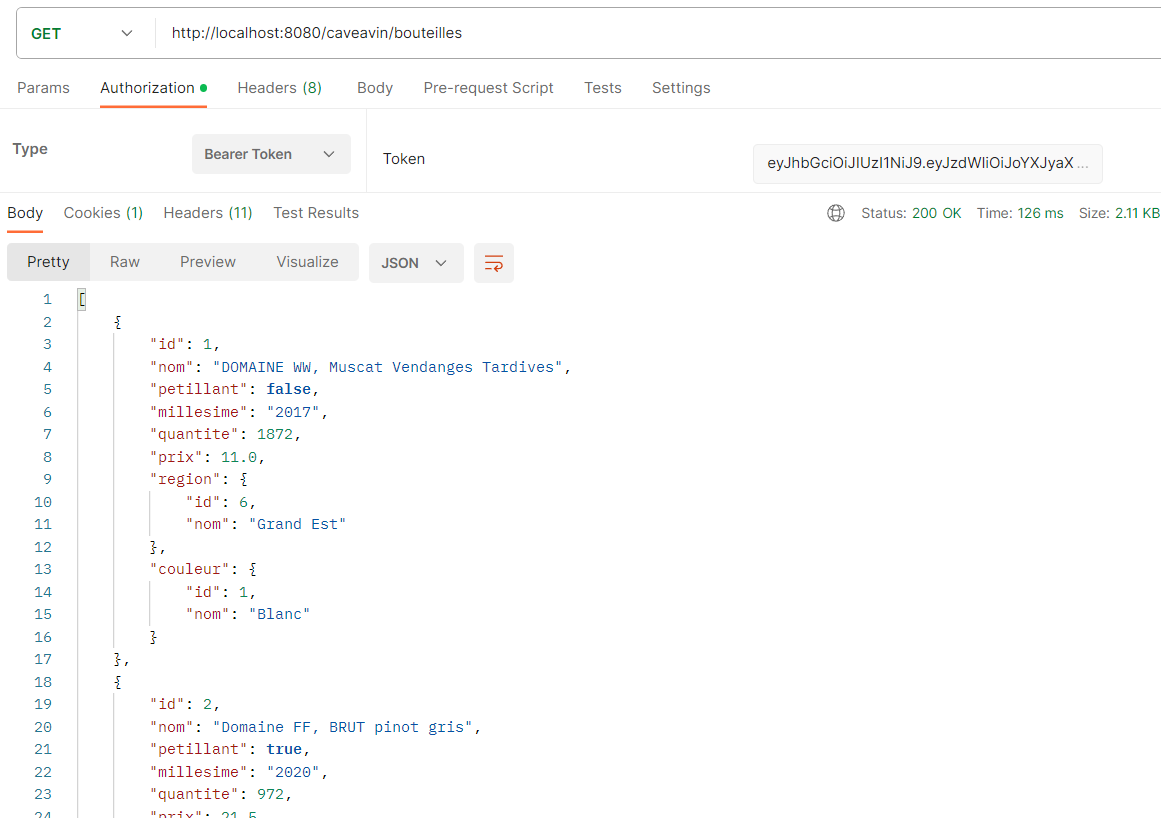


1. Créer un jeton JWT en connectant un Utilisateur avec le rôle OWNER



1. Copier les jetons JWT pour manipuler les URLs de l’application

* Vous devez pouvoir utiliser les 2 jetons pour accéder aux ressources permises par les 2 rôles



* Vous devez avoir un statut 403 si le jeton est associé à un rôle CLIENT alors qu’il veut accéder à des URLs pour OWNER

